LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-635973

Compositional Dataflow Via Abstract Transition Systems

G. Bronevetsky, M. Burke, S. Aananthakrishnan,
J. Zhao, V. Sarkar

May 2, 2013

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Compositional Dataflow via Abstract Transition Systems

Greg Bronevetsky[1] Michael G. Burke[2] Sriram Aananthakrishnan[3]
Jisheng Zhao[2] Vivek Sarkar[2]

[1]Lawrence Livermore National Laboratory, [2]Rice University, [3]University of Utah

Abstract. Despite decades of research and development of dataflow analyses within dozens of different compiler infrastructures, analysis of real programs is intractable in practice. The reason is that although a single research group is equipped to model a single well-defined aspect of application behavior, application developers use a very wide range of abstractions and coding techniques within a single application. Since to analyze even a single application all of its complexities must be modeled, researchers must combine analyses from multiple groups into comprehensive analysis frameworks such as OpenAnalysis or LLVM. Analysis composition is expensive in practice, requiring development effort that is quadratic in the number of analyses as well as tight inter-group coordination to maintain consistent APIs to each other's symbolic abstractions.

This paper proposes an approach to building compiler analysis frameworks that simplifies the composition of independently-developed analyses. It formalizes the operation of dataflow analyses in a way that simplifies reasoning about various types of analysis composition. Further, it defines a portable abstraction that can represent the results of many real analyses, making it possible for different analyses to leverage each other's results **with no knowledge of their APIs or internal abstractions and without any coordination between the groups that developed them**. We have developed the Fuse compositional analysis framework based on this abstraction, and integrated it in the ROSE [15] compilation system. Our experience is that this approach greatly simplifies composition of program analyses, making it easy to tailor different combinations of program analyses to different input programs.

1 Introduction

Composition of multiple analyses is critical for analyzing even simple applications, such as the example in Figure 1. To show that this code must print 10 it is necessary to propagate the constant values of `x` and `y`, eliminate the dead branch of the `if`, propagate the points-to information from the remaining branch to the `print` operation and finally perform more constant propagation to connect the memory location of `*p` with 5 and conclude that `*p+5` is 10. Although all these inferences required for this simple example can be performed by a single monolithic analysis, support for real applications requires many more analyses that require the work of many different developers. Since the complexity of coordinating contributions to a single analysis scales poorly with the number of

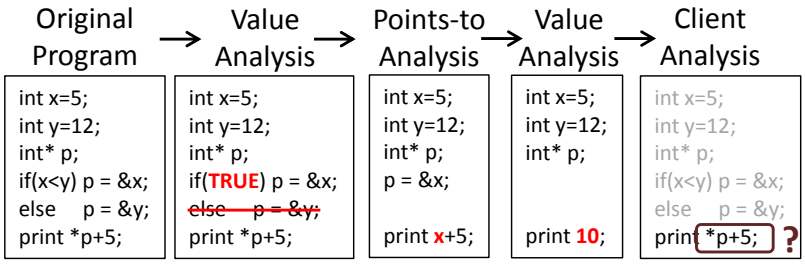

Fig. 1: Application of analysis which chain to example program

developers it is necessary to modularize the overall analysis task into smaller composable analyses that individual developers can focus on.

Composing analyses requires the development of interfaces via which the results of one analysis can be used by another. While this is easy for a few analyses, to enable any analysis to leverage the results of any other requires development work that is quadratic in the number of analyses. Further, these interfaces must be maintained across many versions of each analysis module. The complexity of this approach and its ongoing costs means that it is best suited to well-coordinated compiler development groups that can proactively plan the few analysis pairs that need to be composed and provide long-term support for these couplings.

An alternative approach is to provide common interfaces for classes of analyses, such as the LLVM Alias Analysis interface [2]. While these work well for a few well-established analysis classes, the many different abstractions in common use makes this difficult in many domains. For example, a common interface for analyses that compute constraints on numerical values must account for constants, ranges, linear inequalities and polynomial equalities. Since a comprehensive interface must support very complex functional forms, no such interface has been created in practice. As such, the category of interfaces that exist in practice cover few analyses, as evidenced by the fact that LLVM includes only the Alias Analysis interface, which abstracts the results of just five analyses.

This paper presents a new approach to developing and composing modular compiler analyses based on a single portable formalism that represents the operation of any dataflow analysis and makes it possible to reason uniformly about the inferences made by any analysis. Our work primarily focuses on symbolic dataflow analyses, by which we mean analyses that compute a symbolic over-approximation of the set of possible application executions by symbolically evaluating the application until a fixed point is reached. This includes static dataflow analysis [12], abstract interpretation [4] and symbolic model checking [18]. In this paper all these types of analysis are denoted "dataflow". We do not consider other types of analysis such as graph analyses (e.g. detection or dominators or post-dominators) or property verification.

Our formalism is based on the classic notion of abstract transition systems [4]. Each dataflow analysis is executed on top of one such transition system, associating its own abstract state representation with the transition system's states

and updating it via its own transition relation. In turn, the analysis implements another abstract transition system on which other dataflow analyses may run, using its state representation to specify the states and transitions of this system. We focus on a class of predicates that only maintains information at the granularity of the abstract transition system implemented by their analysis and cannot be used to infer more precise information. This class includes a wide range of analysis, such as classic may- and must-analyses. Each predicate, which denotes a set of executions of a transition system can be be represented portably as an execution of a related transition system where the concrete values, memory locations and operations are replaced with sets of these entities. This transformation restructures the original sets of executions, which are unbounded, into executions of the new system where unbounded-size sets of individual state components are organized in a bounded-size structure. This enables analyses to get opaque objects that denote these unbounded sets and we demonstrate that if these objects implement several key operations arbitrary analyses can use them to perform most analysis tasks. Figure 10 in Section 7 demonstrates how the code sample from Figure 1 can be precisely analyzed using our system.

The main contributions of our work are:

- A formalization of dataflow analyses in terms of executions over abstract transition systems.
- A general abstract representation of analysis results as executions of a set-based transition system that enables composition of multiple analyses.
- A portable encoding of the results of any analysis that is independent of its internal APIs or abstractions.
- A demonstration of the practicality of our approach via a real implementation of the Fuse compositional analysis framework based on our formalism.

2 Summary of Approach

The set of possible application executions can be specified as a transition relation on application states and dataflow analyses attempt to precisely characterize this set. Since in general the set of executions is undecidable, dataflow analyses symbolically over-approximate the true set: all possible executions are in the approximation but some executions it includes are not possible (the approximation is sound but incomplete). This over-approximation is specified in terms of an abstract transition system ("abstraction" for short), where states are disjoint[1] subsets of sub-executions (finitely-long prefixes or suffixes of full executions). There is a transition from abstract state s_1 to state s_2 if there is a transition from a member of s_1 to a member of s_2.

They key idea of composable dataflow is that analyses can both run on abstract transition systems and implement new systems that other analyses will run on. Figure 2 illustrates this by showing a simple code example and four abstractions that over-approximate its set of possible executions. Figure 2(b)

[1] The disjointness condition can be easily relaxed to allow a wider range of analyses, as done with the Trace Partitioning Domain [17]. However, since this significantly complicates the exposition we focus on the disjoint case.

shows a simple abstraction with one state for each control location in the program that denotes all sub-executions that terminate at that location. This is equivalent to a Control Flow Graph(CFG). This abstraction places the constraint that all executions that reach control location C may never return to location A but does not restrict the possible variable values. Figures 2(c) and 2(d) show how the initial CFG abstraction can be refined by additional abstractions that focus on inequalities and parity (Even/Odd) relationships, respectively. The new analyses run on the transition graph of the CFG abstraction and either constrain the set of sub-executions that may exist in each state (e.g. in Figure 2(c) x must be < 10 in state C) and/or partition states into multiple disjoint subsets (e.g. in Figure 2(d) state B is partitioned into sub-states for the cases where y is Even and z is Odd and vice versa). The new constraints remove additional invalid executions from consideration, such as executions that reach state B with $x < 10$ and then immediately terminate execution.

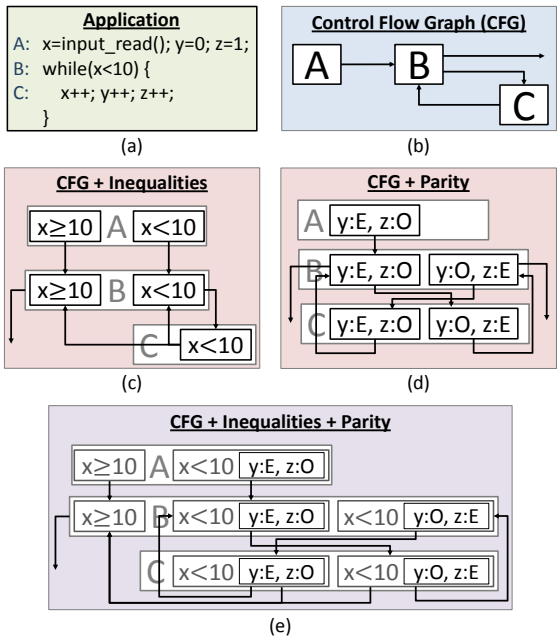

Fig. 2: Sequence of composed analyses

Finally, Figure 2(e) shows the result of composing the three analyses in sequence, first CFG then inequality and finally parity. Each analysis runs on the abstract transition system of its predecessor, refining the sets of the predecessor's states and/or partitioning these states into smaller subsets. As part of their work analyses need to be informed of both the structure of the transition system they run on as well as any information that is known at each state (e.g. the parity analysis may need to know a variable's sign, which the inequality analysis can provide). An analysis is denoted as a "server" if it implements an abstraction for others to run on and a "client" if it runs on an abstraction provided by another analysis. Fore example, in Figure 1, we saw a client analysis at the end of a composition chain which used a preceding value analysis as its server analysis.

The basic concepts of analysis composition are formalized in Sections 3 and 4. Section 3 defines concrete and abstract transition systems. Sections 4 specifies the classic least fixed-point algorithm for computing the solution of dataflow

analyses within this formalism, and shows how the results of dataflow analyses can be used to define the abstractions that other analyses run on.

Each state in an abstract transition system is defined by a logical predicate that carries analysis-generated constraints; Section 5 considers how these predicates serve to denote the sub-executions within their respective states and how their information can be made available to arbitrary analyses. Section 6 demonstrates the utility of this representation by showing how it can be applied in practice to define a portable interface to communicate analysis results. Section 7 then describes the concrete interface of the Fuse compositional analysis framework that we have implemented as part of the ROSE [15] source-to-source compiler. Finally, Section 8 summarizes related work on analysis composition and Section 9 concludes.

3 Transition Systems

We define analysis composition in terms of a formal model of dataflow analysis that is based on prior work on formalizing abstract interpretation [4] and its relationship to dataflow analyses [12] and model checking [18]. It focuses on executions of a concrete transition system and how they relate to executions of an abstract transition system. Our formalization (i) establishes a common set of concepts and terms to describe dataflow analyses, and (ii) define how the results of one analysis can be used by another. Sections 5 and 6 discuss how this information can be communicated via a single portable representation.

We define compositional dataflow by first defining transition systems and the set of their possible executions. We then define the notion of abstract transition systems, or "abstractions", that over-approximate concrete ones by grouping multiple concrete sub-executions into a single abstract state and extending the concrete transition relation to these states. *We then define dataflow analysis as a procedure for computing a more precise abstraction AM (states denote fewer executions) given a less precise abstraction AL by symbolically evaluating AM's transition relation on top of AL's abstract state transition system.* Additional dataflow analyses can then be executed on the AM transition system. Finally, we show how standard notions of dataflow (e.g. may and must analysis) are captured by this definition and apply it to define two types of analysis composition.

A concrete transition system C is defined as $\langle S, S_{init}, S_{fin}, \tau \rangle$. S is the set of all system states, each of which includes information about the current configuration of the computation (defined in more detail in Section 5.1). $S_{init} \subseteq S$ is the set of initial system states, which accounts for all possible application inputs and $S_{fin} \subseteq S$ is the set of final states. $\tau : S \times S$ is the transition relation, written $s \xrightarrow{\tau} s'(s, s' \in S)$, which identifies valid state transitions. Set $AllE$ contains all finite and infinite sequences of states (denoted "executions") in S and $E \subseteq AllE$ contains just the executions that can actually occur. In the general case membership in E is undecidable since transition system C may be Turing-Complete. A "sub-execution" se of some execution $e \in AllE$ is a finite contiguous sub-sequence of e. "Forward" sub-executions start at some state in $s_{init} \in S_{init}$ and continue for a finite number of succeeding states $\langle s_{init}, s_1, ...s_n \rangle$

such that $s_{init} \in S_{init}$ and $s_i \xrightarrow{\tau} s_{i+1}$. Similarly, "Backward" sub-executions end at a state $\in S_{fin}$ and contain a finite number of preceding states. Infinite executions do not have backward sub-executions.

Let $AllSE_{fw}$ and $AllSE_{bw}$ be the sets of all forward and backward sub-executions of executions in $AllE$, respectively. Let SE_{fw} and SE_{bw} be the corresponding sets of sub-executions of the executions in E. We omit the $_{fw}$ and $_{bw}$ where they are clear. An "extension" of a forward sub-execution $\langle s_{init}, ...s_n \rangle$ is a sub-execution $\langle s_{init}, ...s_n, s_{n+1} \rangle$ and conversely for backward sub-executions. $se \stackrel{\tau}{\Rightarrow} se' (se, se' \in AllSE)$ denotes that se' is an extension of se.

An *abstract transition system* $\widehat{A}$ is defined as (the $\widehat{\ }$ symbol identifies entities of an abstract transition system). States $\widehat{se} \in \widehat{SE}$ are disjoint sets of either forward or backward sub-executions such that $SE \subseteq \bigcup_{\widehat{se} \in \widehat{SE}} \widehat{se}$. Let $O(Q)$ denote an over-approximation of set Q ($Q \subseteq O(Q)$). For forward abstractions $\widehat{SE}_{init} = \{\langle s_{init} \rangle \mid s_{init} \in O(S_{init})\}$ and $\widehat{SE}_{fin} = \{\langle s_{init}, ..., s_{fin} \rangle \mid s_{fin} \in O(S_{fin})\}$ and the converse is true for backwards. Transition relation $\widehat{\tau}$ maps one set of sub-executions to the over-approximation of the set of their extensions such that if $se \in \widehat{se} \in \widehat{SE}$ (se is a sub-execution in abstract state $\widehat{se}$) and $\exists se'.\ se \stackrel{\tau}{\Rightarrow} se'$, then

- se' must be contained in some abstract state: $\exists \widehat{se}' \in \widehat{SE}.\ se' \in \widehat{se}'$.
- The abstract transition relation includes the concrete extension: $\widehat{se} \xrightarrow{\widehat{\tau}} \widehat{se}'$.

These conditions ensure that the states of the abstract transition system are closed with respect to the extension relation on sub-executions. Finally, all states in a "forward system" must be reachable from $\widehat{SE}_{init}$ and in a "backward system" they all must reach $\widehat{SE}_{fin}$, both via forward transitions.

4 Dataflow Analysis

Dataflow is a sound but incomplete algorithm for identifying properties that are true of all executions of system C. Soundness ensures that all the discovered properties hold for all executions in E and the incompleteness means that some true properties may be missed. Thus, the set of executions denoted by these properties denotes an over-approximation $\widetilde{E}$ of C's true set E of valid executions (the $\widetilde{\ }$ symbol denotes components and results of dataflow analyses, as shown in Figure 3). The dataflow algorithm, denoted DatA below, assumes the existence of a *finite* abstract transition system $\widehat{A}$ of the concrete system C. For each $\widehat{se} \in \widehat{SE}$ it computes predicate $\widetilde{p}$ that identifies set $[\widetilde{p}]$ such that $SE \cap \widehat{se} \subseteq [\widetilde{p}] \subseteq \widehat{se}$ ($[\widetilde{p}]$ is a tighter over-approximation of the valid sub-executions within $\widehat{se}$). Each set $[\widetilde{p}]$ can be used to implement a state $\widetilde{se}$ of another abstract transition system on which other dataflow analyses execute, as described below. The output of the executed analysis is an abstract state transition system $\widetilde{A} = \langle \widetilde{SE}, \widetilde{SE}_{init}, \widetilde{SE}_{fin}, \widetilde{\tau} \rangle$.

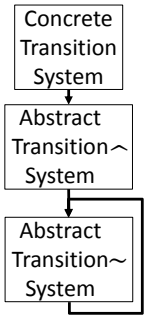

Fig. 3: Dataflow analyses implement abstractions

DatA traverses the states of $\widehat{A}$ using its transition relation $\widehat{\tau}$ to consider sets of increasingly longer concrete sub-executions in $AllSE$. It begins by considering

sub-execution set $\widetilde{SE_0} = \widehat{SE}_{init}$ for forward analyses and $= \widehat{SE}_{fin}$ for backward and its state is initially the set of predicates $\widetilde{P_0}$ such that $SE \cap \widetilde{SE_0} \subseteq [\widetilde{P_0}]$. In each iteration DatA considers extensions of $\widetilde{SE_i}$ under $\widehat{\tau}$ and updates $\widetilde{P}_i$ to ensure that $SE \cap \widetilde{SE_i} \subseteq [\widetilde{P}_i]$ until it reaches a point where $SE \subseteq [\widetilde{P_n}]$. The algorithm operates at the granularity of individual abstract states. For each state $\widehat{se}$ it maintains a separate predicate $\widetilde{p}_i$ and in each iteration extends $\widetilde{SE_i}$ just with extensions along a specific pair of abstract states: $(\widetilde{SE_i} \cap \widehat{se}) \overset{\widehat{\tau}}{\Rightarrow} \widehat{se}'$ using analysis-provided transition function $\widetilde{\tau}$ and merge function $\widetilde{\mu}$. Given abstract states $\widehat{se} \overset{\widehat{\tau}}{\Rightarrow} \widehat{se}'$, $\widetilde{\tau}$ takes the predicate $\widetilde{p}$ at $\widehat{se}$ as well as descriptions of $\widehat{se}$ and $\widehat{se}'$ and computes predicate $\widetilde{p}'$ at $\widehat{se}'$ that contains the extensions of the sub-executions in $[\widetilde{p}]$. $\widetilde{\mu}$ takes in predicates $\widetilde{p}$ and $\widetilde{q}$ and returns predicate $\widetilde{r}$ such that $[\widetilde{p}] \cup [\widetilde{q}] \subseteq [\widetilde{r}]$. DatA combines these to compute predicate $\widetilde{p}'_{i+1}$ at state $\widehat{se}'$ via $\widetilde{p}'_{i+1} = \widetilde{\mu}(\widetilde{\tau}(\widehat{se}, \widehat{se}', \widetilde{p}_i), \widetilde{p}'_i)$. The algorithm thus maintains the above invariant at the granularity of each abstract state: $\forall i, \widehat{se} \in \widehat{SE}.\ SE \cap \widetilde{SE_i} \cap \widehat{se} \subseteq [\widetilde{p}_i]$. Because the size of the encodings of the predicates is bounded, the number of different predicates must also be bounded. As such the algorithm must reach a point where $\widetilde{P}_{i+1} = \widetilde{P}_i$, meaning that $[\widetilde{P}_i]$ already includes the next extension of $\widetilde{SE_i}$ and thus all of its subsequent extensions.

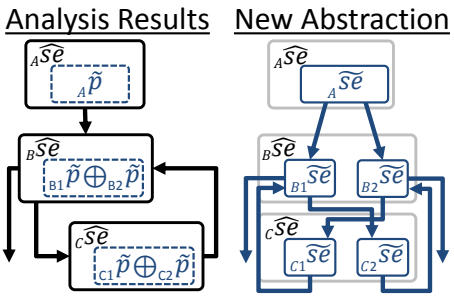

Fig. 4: Implementation of abstraction.

The predicates computed by an analysis at each state of abstraction $\widehat{A}$ can be used to implement another abstraction $\widetilde{A}$. This is illustrated in Figure 4, which shows the computation of the CFG+Parity abstraction from Figure 2(d). The left sub-figure shows states ${}_x\widehat{se}$ of the CFG abstraction, with their associated predicates ${}_x\widetilde{p}$ computed by the Parity analysis. The right one shows the abstraction defined by these predicates, with states ${}_x\widetilde{se}$. Predicate ${}_A\widetilde{p}$ constrains the sub-executions in state ${}_A\widehat{se}$ to those where y is even and z is odd and denotes a single state in $\widetilde{A}$. Predicates at states ${}_B\widehat{se}$ and ${}_C\widehat{se}$ can be decomposed into two disjoint cases: "y is even, z is odd" and vice versa. Each case implements a separate state of abstraction $\widetilde{A}$. $\widetilde{A}$ denotes fewer executions than $\widehat{A}$ because: (i) individual states in $\widetilde{A}$ denote fewer sub-executions than their counterparts in $\widehat{A}$ and (ii) the splitting of states in $\widehat{A}$ causes some state transitions that were present in $\widehat{A}$ to be excluded from $\widetilde{A}$ (i.e. the edges are sparser).

Types of Composition Consider the execution of transition function $\widetilde{\tau}$ of some analysis on transition $\widetilde{se}^S \overset{\widetilde{\tau}^S}{\Rightarrow} \widetilde{se}'^S$ of an abstraction implemented by server analysis S. $\widetilde{\tau}$ takes as input predicates P and P' that describe states $\widetilde{se}^S$ and $\widetilde{se}'^S$ and uses them, along with $\widetilde{\mu}$, to compute $\widetilde{p}'_{i+1}$. This predicate over-approximates the set of valid sub-executions in state $\widetilde{se}'^S$ that have been considered up to iteration $i+1$: $SE \cap \widetilde{SE_{i+1}} \cap \widetilde{se}'^S \subseteq [\widetilde{p}'_{i+1}]$. The descriptions P and P' must

thus over-approximate the valid sub-executions in $\widetilde{se}^S$ and $\widetilde{se}'^S$ that have been considered up to the preceding iteration i.

A simple choice is $P = \widetilde{p}^S$ and $P' = \widetilde{p}'^S$, since predicates $\widetilde{p}^S$ and $\widetilde{p}'^S$ correspond to all sub-executions: $SE \cap \widetilde{se}'^S \subseteq [\widetilde{p}]$ and same for $\widetilde{p}'^S$. This is denoted *loose* composition and corresponds to one analysis executing to completion before its predicates are used by other analyses. However, suppose that multiple client analyses C_1, ..., C_n are running on the same abstraction, their transfer and merge functions executed concurrently in each iteration. In this case, in iteration i predicate $\widetilde{p}_i^{C_j}$ computed by analysis C_j over-approximates the valid executions in state $\widehat{se}$ that have been considered at i. Thus, the transition function $\widetilde{\tau}^{C_k}$ of analysis C_k can choose $P = \widetilde{p}_i^{C_j}$ $(j \neq k)$ and same for P'. Further, since the intersection of over-approximations is itself an over-approximation, an even better choice is $P = \bigwedge_{l \neq k} \widetilde{p}_i^{C_l}$ and $P' = \bigwedge_{l \neq k} \widetilde{p}_i'^{C_l}$. This approach, denoted *tight* composition, was performed manually by Wegman and Zadeck [19] to combine constant propagation and unreachable path elimination. Lerner, et al. [10] develop an analysis framework that automates tight composition and show that tight composition is more powerful than loose for programs with loops.

Application of Framework to Analysis Types We now discuss how traditional types of analysis fit into our framework. In may-analyses, predicates describe the disjunction of facts that may hold on some of the sub-executions within a given abstract state. As longer sub-executions are considered the merge function $\widetilde{\mu}$ adds more disjuncts to include their information.Thus, as more sub-executions are added to $\widetilde{SE}_i$, the disjuncts expand set $[\widetilde{p}]$ at each abstract state $\widehat{se}$ to ensure that $SE \cap \widetilde{SE}_i \cap \widehat{se} \subseteq [\widetilde{p}]$ always holds. In must-analyses, predicates are conjunctions of facts that must hold for all the sub-executions within an abstract state. The predicate at each abstract state is a conjunction of term. As more sub-executions are considered $\widetilde{\tau}$ computes the conjuncts that describe them and $\widetilde{\mu}$ removes conjuncts as needed to ensure that the remaining ones hold for all sub-executions that have been considered. As conjuncts are dropped, the sets denoted by each predicate grow larger, always maintaining the above condition.

Optimistic and pessimistic analyses [19] offer a different tradeoff in analysis design. Optimistic analysis initializes every non-starting abstract state $\widehat{se}$ with a predicate $\widetilde{p}$ such that $[\widetilde{p}]$ under-approximates $SE \cap \widehat{se}$, usually just $[\widetilde{p}] = \emptyset$. Then, as more sub-executions are considered during analysis, $\widetilde{p}$ is expanded to account for them all, ultimately reaching an over-approximation of $SE \cap \widehat{se}$. In contrast, pessimistic analyses initialize $\widetilde{p}$ to be an over-approximation of $SE \cap \widehat{se}$, usually $[\widetilde{p}] = \widehat{se}$. While the more aggressive assumptions of optimistic analyses make them more accurate, pessimistic analyses converge more quickly and produce conservative results even if they are terminated prematurely.

5 General Abstract Representation of Analysis Results

Section 4 established a common framework for reasoning about how dataflow analyses operate and how they can compose. A server analysis provides an abstract transition system on which the transition and merge functions of client analyses run. These functions query from the server or concurrently-executing

clients descriptions of the predicates that define the abstract states on which they run. This section describes a portable interface for describing these predicates that enables generic composition of a large class of analyses. It begins by defining a general concrete transition system that captures the key properties of Turing-Complete transition systems, and then formalizes the notion of "Abstraction Granularity" predicates in terms of the information that can be observed about the sub-execution sets they denote. It then shows that the observable information about the individual state components within these sub-executions induces a state transition system that is very closely related to the original transition system. Sub-executions of this transition system form a compact portable representation of the information content of these predicates.

We focus on predicates $\widetilde{p}$ that provide information at the granularity of the abstract transition system implemented by their analysis and cannot be used to infer more precise information about the sub-executions within each abstract state. In other words, all the information that can be queried about the predicates by other analyses is encoded in the structure of the abstract transition system the predicates denote and there is no additional information hidden in their internal encodings. Specifically, *Abstraction Granularity* (AG) means that for any query function f that may be applied to the set of sub-executions $[\widetilde{p}]$ induced by predicate $\widetilde{p}$, it must be true that $f([\widetilde{p}]) = \bigcup_{se \in [\widetilde{p}]} f(\{se\})$. Since the information returned on the entire set is the union of the information returned on its individual sub-executions, the different sub-executions cannot be distinguished from each other and as Section 5.2 discusses in more detail, the same must be true of their individual states and their internal components. For example, let set $[\widetilde{p}]$ contain two sub-executions and let s and s' be their final states. s computes $1+2$ and s' computes 5×2. A query of the operation performed by the last states in $[\widetilde{p}]$ would return set $\{+, \times\}$, while queries of their inputs would return $\{1, 5\}$ and $\{2\}$. Finally, a query of the output of the computations would return the set $\{3 = 1+2, 7 = 5+2, 10 = 5 \times 2, 2 = 1 \times 2\}$, which includes all the permutations of operation and input values to maintain consistency among all the results of all queries. The observable properties of sub-execution sets denoted by AG predicates act like a single sub-execution of a variant of the concrete transition system where the concrete values, memory locations and operations are replaced by sets of these entities. This close relationship between the concrete and the AG transition systems makes it possible to use the AG system as a portable and easy-to-use representation for accessing the information provided by AG predicates from arbitrary analyses.

Section 5.1 presents a generic formalization of the structure of individual transition system states that captures the key properties of Turing-Complete transition systems. Section 5.2 then discusses the limits on the information available about the sub-executions denoted by AG predicates, as well as their states and internal state components. It then presents a way to convert the original transition system into a variant that simulates the observable behavior of sub-executions denoted by AG predicates.

5.1 Structure of Concrete States and Sub-Executions

We now define the states of concrete transition system C (defined in Section 3) as a generic state structure that can be adapted to represent any real-world transition system. Let state $s \in S = \langle \sigma \in Store, \kappa \in Comp \rangle$, where

- $Store = MemLoc \mapsto Data$: set of mappings from memory locations to their contents, where
  - $Data = Value + MemLoc + Operation$: set of data kept in the store.
  - $Value$: set of data items, including scalars, arrays and classes.
  - $MemLoc$: set of memory locations.
  - $Operation$: set of operations that may be performed.
- $Comp = (Operation \times MemLoc \times (MemLoc \times ...)) \mapsto (Data \times Comp)$: set of computations, which include an operation, the memory location where it will store its results as well a finite tuple of the $MemLocs$ where its operands are stored. Computations return a pair with the $Data$ to be written to the output memory location and the $Comp$ of the next state.

The store and computation of state s are denoted $s.\sigma$ and $s.\kappa$, respectively. For any $ml \in MemLoc$ $s.\sigma[ml]$ denotes the $Data$ element stored at ml. The individual fields of $s.\kappa$ are denoted as follows: (i) $s.\kappa.op$ is the $Operation$ field, (ii) $s.\kappa.outML$ is the output $MemLoc$, (iii) $s.\kappa.inML_0$, ..., $s.\kappa.inML_n$ denote all the input $MemLocs$, (iv) $s.\kappa.outData$ is the output $Data$, and (v) $s.\kappa.outComp$ is the output $Comp$. Further, each element $data$ of set $Data$ (i.e. element of set $Value$, $MemLoc$ or $Operation$) is denoted a *state component* since all larger state structures are built from these units.

This state structure is very general and can be instantiated to represent an imperative language or the λ calculus, depending on the definition of the transition relation τ, the choice of operations and details such as whether $MemLocs$ are typed or untyped and whether the number of code locations is finite (e.g. lines of code in an imperative program) or unbounded (e.g. λ calculus or Forth [8], which generate computations dynamically).

5.2 Abstraction Granularity

Sub-Execution Sets and Their Visible Behaviors Recall that we are focused on predicates $\widetilde{p}$ that have Abstraction Granularity(AG): they only provide information at the granularity of the abstract transition system implemented by their analysis. Formally, consider all the abstract states $\widetilde{se} \in \widetilde{SE}$ implemented by some analysis and the predicates $\widetilde{p}$ that denote them ($[\widetilde{p}] = \widetilde{se}$). For any query function f that takes sets of sub-executions as input it must be true that $f([\widetilde{p}]) = \bigcup_{se \in [\widetilde{p}]} f(\{se\})$. A concrete sub-execution is a sequence of states and each state is constructed from components, each of which is an element of set $Data$. As such, function f may inspect any component(s) of any state(s) in any sub-execution $\in [\widetilde{p}]$ and return just these components or a function of them (e.g. set of the sum of an array's entries in the last state of all sub-executions $\in [\widetilde{p}]$). To ensure that the results of all such query functions are consistent, the above constraint must apply to individual states within sub-executions and the individual components of those states. For example, let $se.last$ denote the last state of a

sub-execution se. If $se \in [\widetilde{p}]$ and $f(\{se\}) = se.last.\sigma[m]$ for some $m \in MemLoc$ then $f([\widetilde{p}]) = \{se'.last.\sigma[m] \mid se' \in [\widetilde{p}]\}$.

By definition the result of every query on an abstract state denoted by an AG predicate must be a set. This means that from the perspective of outside observers the sub-executions in $[\widetilde{p}]$ are sequences of sets of states that are constructed from sets of state components. Figure 5 illustrates the relationship between the concrete sub-executions $\in [\widetilde{p}]$ and their observable behavior, based on the code in Figure 2. On the left the figure shows the code and the CFG abstraction for this code, which has three states, A, B and C. The top-right shows two representative full executions, which contain multiple forward sub-executions. All the sub-executions that terminate in a column are members of the corresponding abstract state. The bottom-right of the figure shows this abstraction's visible behaviors. For instance, if query function f reads the value computed by $Operation <$ in the last state of any sub-execution $\in B$ then it must return set $\{\text{True}, \text{False}\}$, which is all the $Values$ computed in all the last states. If f reads the value stored at $MemLoc$ z in the last state of an sub-execution $\in C$, it must return $\{1, 2, ...\}$. This is true of any function that accesses state components or more complex functions over multiple components or states.

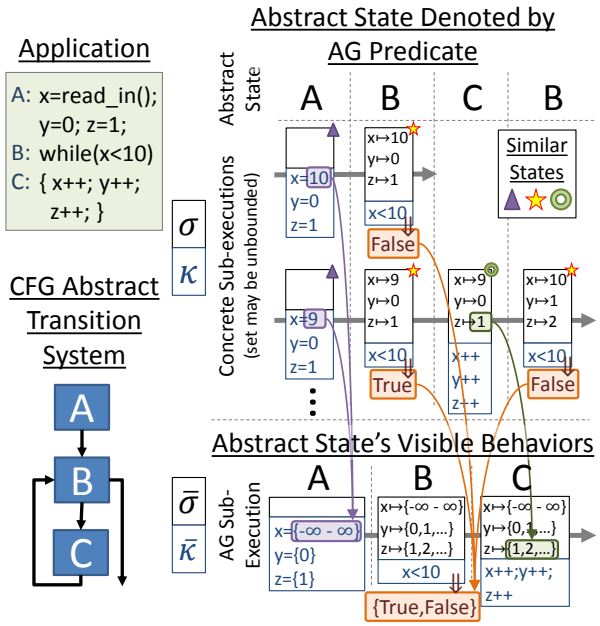

Fig. 5: Observable behavior of sub-execution sets denoted by AG predicates

Since sub-executions in each abstract state have variable lengths we must define what is meant by a query reading the components of "some specific state" in a sub-execution set. We define state identity at the granularity of the abstract transition system. Concrete states s and s' are considered *similar* if they are the last states of concrete forward sub-executions se and $se' \in \widetilde{se}$ (first state for backward sub-executions). The three sets of similar states are marked in Figure 5 with a different symbol. The result of a query on some state s is the union of the results it returns on all states similar to s.

AG Transition Systems The key insight is that the observable behaviors of set $[\widetilde{p}]$, denoted by AG predicate $\widetilde{p}$, can be formalized as another state transition system, denoted the *AG transition system* such that a single sub-execution of

this system denotes all the observable behaviors of $[\widetilde{p}]$. This is implied directly from the requirement that the results of queries on $[\widetilde{p}]$ must be consistent: a query on the outputs of an operation must return a set consistent with the execution of the operation on the subsets of $Data$ returned by queries of its inputs, which in turn must be consistent with the state of the store. The dependences among query results induce a state transition system that is structurally very similar to the concrete transition system, making it easy for analyses to read and create.

State Structure: AG states are defined such that each denotes the observable properties of an equivalence class of similar concrete states. Given AG state $\bar{s}$, the corresponding set of similar concrete states is $[\bar{s}]$ (symbol $\bar{}$ identifies AG states and associated entities such as sub-executions and state components). The results of all queries on set $[\widetilde{p}]$ decompose into reads of individual state components of one or more of its state similarity sets and must return subsets of sets $Value$, $MemLoc$ or $Operation$. To model this AG states are built from subsets of these sets and all fields of AG states such as the store σ and computation κ are constructed from these subsets. Formally, the states of $\overline{AG}$ are defined as $\overline{S} = \langle \overline{\sigma} \in \overline{Store}, \overline{\kappa} \in \overline{Comp} \rangle$, where

– $\overline{Store} = (\mathcal{P}(MemLoc) \times \mathcal{P}(Data)) \times \overline{Store}$

– $\overline{Comp} = (\mathcal{P}(Operation) \times \mathcal{P}(MemLoc) \times (\mathcal{P}(Data) \times ...)) \mapsto (\mathcal{P}(Data) \times \overline{Comp})$

$\overline{S}$ is identical to the definition of concrete system states S, except that (i) all instances of sets $Value$, $MemLoc$ and $Operation$ are replaced with their power sets and (ii) the store is modeled as an ordered list of $\langle \mathcal{P}(MemLoc) \mapsto \mathcal{P}(Data) \rangle$ pairs rather than a function from $MemLocs$ to $Data$ elements, for reasons described below. Like above, subsets of $Data$ (i.e. an element of set $\mathcal{P}(Value)$, $\mathcal{P}(MemLoc)$ or $\mathcal{P}(Operation)$) are denoted *AG components*.

State and Transition Semantics: First, we define the semantics of an individual AG state. Given AG state $\bar{s}$ consider the queries that read the fields of concrete states $\in [\bar{s}]$. A query that reads field $\kappa.op$ of all these states must produce the set $\{s.\kappa.op \mid s \in \bar{s}\} \subseteq Operation$. $\bar{s}.\kappa.op$ is thus defined to be this set. The field $\bar{s}.\sigma$ and the other sub-fields of $\bar{s}.\kappa$ are defined the same way.

We now define the semantics of AG state transitions. Given AG states $\bar{s}$ and $\bar{s}'$, let $Trans$ be the set of concrete state transitions from members of $[\bar{s}]$ to members of $[\bar{s}']$: $\{\langle s \in [\bar{s}], s' \in [\bar{s}'] \rangle \mid s \xrightarrow{\tau} s'\}$. If $Trans \neq \emptyset$ we define an AG state transition $\bar{s} \xrightarrow{\bar{\tau}} \bar{s}'$. AG state $\bar{s}'$ must be consistent with the execution of $Comp\ s.\kappa$ in each pair $\langle s, s' \rangle \in Trans$. The execution of $s.\kappa$ affects s' in two ways: (i) $s'.\kappa = s.\kappa.outComp$ and (ii) $s'.\sigma[s.\kappa.outML]$ is mapped to $s.\kappa.outData$. To capture this we must first ensure that the outputs $\bar{s}.\kappa.outData$ and $\bar{s}.\kappa.outComp$ are valid. From the perspective of queries, $\bar{s}.\kappa.op = \bigcup_{s \in \bar{s}} s.\kappa.op$ and each $Operation$ in set $\bar{s}.\kappa.op$ may read as input any element $\in \bar{s}.\sigma[\bar{s}.\kappa.inML_i] = \bigcup_{s_1, s_2 \in \bar{s}} s_1.\sigma[s_2.\kappa.inML_i]$. To maintain consistency $\bar{s}.\kappa.outData$ and $\bar{s}.\kappa.outComp$ must be the sets of the outputs of all of the above operations running on all of the above inputs: $\{op(..., inData_i, ...) \mid op \in \bar{s}.\kappa.op, inData_i \in \bar{s}.\kappa.inData_i\}$. Further, the results $\bar{s}.\kappa.outData$ and $\bar{s}.\kappa.outComp$ must be written to $\bar{s}'$: (i) $\bar{s}'.\sigma[\bar{s}.\kappa.outML] \supseteq \bar{s}.\kappa.outData$ and (ii) $\bar{s}'.\kappa \supseteq \bar{s}.\kappa.outData$ The relation is a superset rather than equality since there may be transitions to some concrete states $\in [\bar{s}']$ from con-

crete states outside of $[\bar{s}]$, which may add additional elements of $Comp$ and $Data$ to $\bar{s}'.\sigma[\bar{s}.\kappa.outML]$ and $\bar{s}'.\kappa$, respectively.

We now define the AG store. The mapping of some $\overline{ml} \subseteq MemLoc$ in store $\bar{s}.\sigma[\overline{ml}]$ is $\overline{data} \subseteq Data$ such that elements $data \in \overline{data}$ were written to some $ml \in \overline{ml}$ earlier in its respective sub-execution and not subsequently overwritten. The contents of $\overline{data}$ can be computed by considering the assignments in AG states that precede $\bar{s}$ in $[\bar{p}]$, from newest to oldest. If we reach state $\bar{s}_j$ and observe an assignment of $\overline{ml}_j \subseteq MemLoc$ to $\overline{data}_j \subseteq Data$ we add $\overline{data}_j$ to $\overline{data}$ and remove $\overline{ml}_j$ from consideration. This continues until all elements in $\overline{ml}$ have been eliminated. This definition means that AG stores must be structured as an ordered list of assignments $\langle \overline{ml}_j \mapsto \overline{data}_j \rangle$. The first pair denotes the assignment performed in the AG state that immediately precedes $\in \bar{s}$ in the sub-executions in $[\bar{p}]$. The next pair denotes the state that precedes that one, etc.

Reviewing, we have presented AG predicates and showed how the results of queries on them can be modeled as a single sub-execution of an AG transition system. This transformation replaces the *set* (possibly unbounded) of concrete sub-executions in an abstract state with a *single* AG sub-execution, where every AG state component is a set (possibly unbounded) of elements in $Value$, $MemLoc$ and $Operation$. By organizing abstract state descriptions with the bounded-size aspects (structure of AG states) at the highest level and the unbounded aspects (contents of AG components) at a lower level, we make them accessible by client analyses at analysis-time. Further, by wrapping AG components behind an interface we enable them to be transparently combined and related via appropriate implementations of interface methods.

6 Compositional Analysis Interface

This section defines a portable interface to communicate analysis results that allows clients to give servers a source code expression and ask for opaque objects that denote the sets of values, memory locations or operations this expression evaluates to. Each object (i) over-approximates the sets of elements the its source expression denotes given and (ii) implements basic set-related methods. We show that this is sufficient to enable clients to (i) portably infer new information and (ii) build Abstraction Granularity predicates to represent their own information.

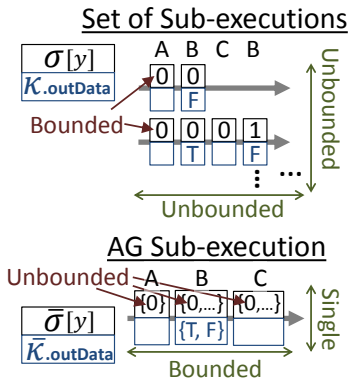

Fig. 6: AG predicates as sub-execution sets or a single AG sub-execution

The AG transition system defined in Section 5.2 is a useful normal form for analysis results because it describes AG predicates not as sets of sub-executions of the concrete system but as a single AG sub-execution, built from sets of concrete state components. Figure 6 illustrates this difference for the code in Figure 2, focusing on queries of the $\kappa.outData$ field and $\sigma[y]$ in the last state of every sub-execution $\in [\widetilde{p}]$. The raw set $[\widetilde{p}]$ is portable across all analyses because it is

expressed purely in terms of the concrete transition system. However, because its size may be unbounded, its contents cannot be directly enumerated. This makes it impossible for client analyses to construct the set of all $\kappa.outData$ instances. In contrast, because the high-level structure of AG sub-execution is bounded in size, clients can directly access the last AG state $\bar{s}$ and its $\bar{s}.\kappa.outData$ field in bounded time.

The AG components returned by such queries are opaque objects that (i) cannot be directly inspected and (ii) denote potentially unbounded sets that cannot be enumerated in general (we denote such objects the *representations* of AG components). Below we show how to define the interface that must be provided by representations to support the two key use cases. First, clients need to create representations of their information at each abstract state in terms of an AG sub-execution. Since no analysis can accurately model all aspects of an application on its own, clients need to federate the AG component representations they compute with those computed by multiple other analyses into a single representation of an AG sub-execution. Second, clients need to make inferences about possible sub-executions to make decisions about the structure of the AG sub-executions they create. Section 6.1 discusses how AG components are created and how some of their properties can be determined portably. Section 6.2 then considers the operations that analyses must implement in their AG component representations to enable clients to combine them into their own AG sub-execution representations.

6.1 Creation of AG Components

Client analyses create AG component representations for their use in two ways. First, they can query specific components of AG states and receive the corresponding representations computed by different analyses. Each representation will denote a different set of concrete state components since each analysis computes a different approximation. This supports clients that need information about specific portions of the application but not clients that can perform simplifying transformations. For example, if it is known that `a=x+y` and `b=x-y`, then expression `a-b` can be replaced with expression `2*y`. A client that performs this replacement still needs to query the AG components that denotes the values of expressions `2` and `y` and/or `2*y`, even though they do not exist in the original code. As such, it must be possible to create representations for AG components that exist in original sub-executions and ones that are newly generated.

Although in general AG components cannot be enumerated due to their unbounded size, there are ways for clients to learn more about them. First, since equality relations *may* = and *must* = are evaluated at analysis time and return True or False, analyses can use them to create complex structures such as the ordered list representation of AG $\overline{Stores}$ described in Section 5.2. Further, in many cases AG components are actually finite in size. For example, the result of any boolean operation is known to be either True or False and in Figure 1 the set of values printed is just $\{10\}$. The contents of such AG components can be enumerated and used to make compile-time decisions. Further, analyses can probe the properties of AG components by creating expressions that involve

them and querying the AG components that denote these expressions' values. For instance, the possible values of variables `x` and `y` can be compared by taking their AG *Data* component representations and querying the AG *Data* component of expression `x < y`. Since this AG *Data* component is finite, it can be fully enumerated to to determine whether it is {True}, {False} or {True, False}.

6.2 Interface to Enable Clients to Create AG Sub-executions

We now focus on creating an interface that enables analyses to create the AG sub-executions that define their abstract states. Figure 7 shows the key pieces of an AG sub-execution, focusing on a single AG state and how it connects to the following AG state. AG sub-executions are built by connecting individual AG components by (i) creating tuples of AG components and (ii) connecting the inputs of an AG *Operation* component to its outputs. First, to create usable tuples of AG component representations, these representations must support a notion of equality. Figure 8 lists two set relational operations that approximately determine whether the contents of these AG component sets are equal. Second, given transition $\bar{s} \xrightarrow{\bar{\tau}} \bar{s}'$ it must be possible to connect the results of computation $\bar{s}.\kappa$ to $\bar{s}'.\sigma$ and $\bar{s}'.\kappa$. Analyses must thus provide functions that simulate each $\bar{s}.\kappa.op$. These must take as input AG *Data* components for each $\bar{s}.\kappa.InData_i$ and return AG *Data* and *Computation* components for $\bar{s}.\kappa.outData$ and $\bar{s}.\kappa.outComp$, respectively.

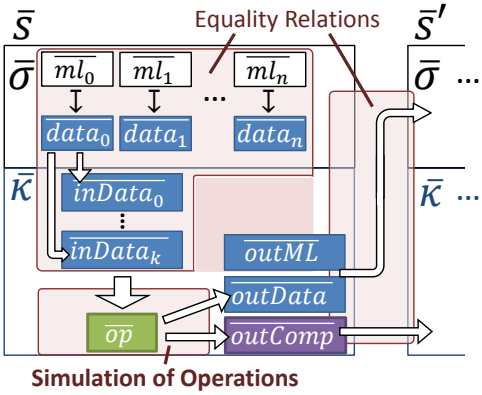

Fig. 7: AG sub-execution structure in terms of the constituent AG components and the operations they implement.

Combining AG Components from a Single Analysis Given only AG component representations implemented by a single analysis, implementing the above functions is simple because the internal structure of each representation is available for inspection. For example, consider constant folding, such as might take place within constant propagation analysis, where AG *Data* components may be in states (i) empty set (denotes uninitialized data), (ii) known constant and (iii) universal set (more than one possible value). Equality operations are implemented by applying the definitions in Figure 8 to the above sets. *Operations* are simulated as follows. Let $\overline{data}$ and $\overline{data}' \subseteq Data$ be two AG *Data* components. If both are in state known constant, the operation is applied to these constants and an AG component with that constant value is returned. Otherwise, it returns universal set or empty set following the standard semantics of constant folding expression evaluation. This functionality can be implemented by any analysis and usually already exists in their transition functions.

Combining AG Components from Multiple Analyses In practice clients need to combine AG components from multiple analyses to create a representation of their own AG sub-executions. This is significantly more complex because representations from different analyses know nothing about each other's internal state. This means that in general to accurately compute set equality operations or simulate *Operations* it is necessary to write implementations specialized to the representation at hand.

a **may** $= b$ if $a \cap b \neq \emptyset$
a **must** $= b$ if a *may* $= b$ and $\lvert a \rvert = \lvert \overline{s'} \rvert = 1$

Fig. 8: Equality operations on sets a, b

Fortunately, for many common scenarios it is possible to leverage multiple analyses without requiring them to interact directly. For example, to compute the AG $Value$ component for the result of expression `x + y` we can ask multiple analyses to concurrently provide AG representations for it. The vector of these representations can implement a new AG component, where $op \in Operation$ is simulated by calling the op simulator of each analysis element-wise on its own AG component representations and same for equality. Since each representations over-approximates the real set of $Values$ in `x+y`, their intersection is also an over-approximation. Thus, it is legal to respond to any function call with results from the representation that returns the most precise result. Specifically, two vectors are $must =$ to each other if any of their elements are $must =$ to each other. Similarly, if any representations in a vector are finite sets, their intersection can be enumerated even if other representations are not finite. This mechanism, denoted "parallel composition" in our implementation in Section 7, enables clients to simultaneously utilize multiple server analyses, automatically leveraging the best one for each situation.

Communicating Structures of AG Components Structures larger than individual AG components can be communicated in terms of individual AG components. While simple for bounded structures like $\overline{Computations}$, this is complex for unbounded structures such as $\overline{Stores}$, arrays or invariants such as "variable is not assigned during a given loop". To compose such structures it is necessary to walk an AG sub-execution or look at the last AG state of multiple abstract states. For example, AG $\overline{Stores}$ are specified in Section 5.2 as an ordered list of assignments, each performed in a different AG state. An analysis can create the same representation incrementally during the dataflow analysis. When its transition function $\widetilde{\tau}$ is called on an abstract state where $\bar{s}$ is the last AG state, $\widetilde{\tau}$ can add the mapping $\langle \bar{s}.outML \mapsto \bar{s}.outData \rangle$ to its representation of the store until a fixed point is reached. Loop invariants can be identified in the same way by checking in $\widetilde{\tau}$ whether $\langle \bar{s}.outML$ is the variable in question. This approach is functional but still fairly complex to use. In our future work we will focus on compact representations for complex AG structures such as memory regions, which will enable portable representations of entities such as arrays, lists or graphs using a common interface.

7 Implementation

We have use the above formalism to develop the Fuse compositional analysis framework as part of the ROSE [15] source-to-source compiler. Fuse is focused on

analysis for imperative programming languages, with active support for C/C++ and plans for supporting other imperative languages such as Fortran and Java. We have used Fuse to implement and compose the following analyses: liveness detection, dead-path elimination, array index analysis, constant propagation, points-to analysis and value numbering. Each analysis was developed independently and does not rely on any details of the other analyses, even though the first four operate on dense control-flow graphs, while the last three are operate on sparse SSA [6] graphs (there are two versions of constant propagation, a dense and a sparse).

Framework Structure Fuse is organized around a "Composer" and multiple "Analyses". Each analysis provides a transition function and a representation of AG sub-executions. It is executed by running its transition function on states of an abstraction implemented by another analysis and implements an abstraction on which other analyses may run. The analysis implements the AG sub-execution representation that defines each state of its abstraction. Figure 9 shows how the composer mediates interactions between analyses. It chains analyses in series or in parallel, forwarding any function calls that query abstraction properties to the analysis that implements them (each analysis may implement only a portion of abstraction functionality, as discussed below). The composer begins by executing a non-dataflow base analysis that uses information from the raw abstract syntax tree (e.g. control-flow graph and variable types) to implement an abstraction. Subsequent analyses run on this abstraction.

Analyses implement an API that specifies both the abstraction graph structure and each state's AG sub-execution. Methods `getStart` and `getEnd` return the graph nodes for the application's entry and exit points. These nodes implement methods that return their successors and predecessors, allowing callers to walk the graph. In particular, the implementation of this API in the base analysis exports the classic control flow graph.

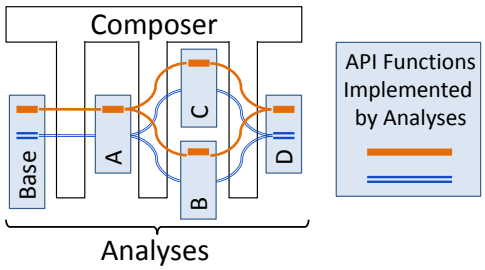

Fig. 9: Structure of Fuse

Analyses implement methods to query the structure of the AG sub-execution that denotes each abstract state. To reduce the complexity of our interface only allows access to the AG components of the terminal AG state of the sub-execution (last for forwards analyses and first for backwards). Clients get the AG component representation that denotes a given expression by calling methods `Expr2Value`, `Expr2MemLoc` and `Expr2Operation`, which are forwarded by the composer to one or more analyses. These methods return an opaque object of type `AGValue`, `AGMemLoc` and `AGOperation`, respectively, that represent an AG component of the terminal AG state and implement methods `mayEqual` and `mustEqual`. Clients simulate the results of operations on

AG component representations by calling `Expr2*` on expressions they create (e.g. call `Expr2Value(a+b)` to simulate `+`). Finally, each AG component representation implements method `isFinite` that returns whether it denotes a finite set. If so, method `getConcrete` returns the components in the set.

To enable analysis developers to focus on their domain of expertise, analyses may implement only a portion of our interface. Graph access methods can be implemented without implementing the `Expr2*` methods and each `Expr2*` method can be implemented without implementing the others. As Figure 9 shows, the composer detects missing implementations and finds the analysis to service each call. Importantly, each `Expr2*` implemented method must return a valid object for all expressions passed as arguments. This ensures that all `mayEqual`, `mustEqual` calls and the simulation of almost every operation must operate on objects from the same implementation (except pointer arithmetic, which uses an `AGMemLoc` and an `AGValue` as arguments).

Fuse currently supports sequential and parallel loose composition. In loose composition, defined in Section 4, each analysis must run to completion before another analysis may use its results. Support for tight composition (also known as super-analysis [19,10]), where analyses are executed and can communicate concurrently, is part of our future work. Under sequential composition analyses that execute later are run on abstract transition systems implemented by prior analyses and is exemplified by analyses `Base` and `A` in Figure 9. Under parallel composition analyses execute independently on the same abstraction. All API calls by subsequent analyses are forwarded to all the analyses composed in parallel and the AG component representation that is returned is a vector of their results. Analyses `B` and `C` in Figure 9 are composed in parallel. As discussed in Section 6.2 such vector AG component representations denote the intersection of their members since the result of any operation on the vector is the most precise result returned by any member representation.

Case Study We now show how the example in Figure 1 can be analyzed precisely by composing the following analyses already implemented in Fuse : Constant Propagation, Unreachable Path Elimination and Points-To Analysis. The composition and how it computes the result are shown in Figure 10.

The composer first executes the Base analysis, which creates an initial abstract transition system from the ROSE Virtual CFG and type information. Each abstract state contains all sub-executions that end at (forwards sub-executions) or start at (backwards sub-execution) a given code location. The `AGMemLoc` objects it implements model lexical variables precisely but capture no detail on heap memory or memory accessed through pointers (all these $MemLocs$ are $may =$ to each other). Similarly, while `AGMemValues` for literal constants denote a set with just that constant, `AGValues` for memory contents denote the universal set. `AGOperations` are fully precise except for function pointers. Base implements the full compositional API: abstraction graphs and each `Expr2*` function.

Base is followed by a scalar Constant Propagation (CP) analysis, which has the functionality of the classic Kildall algorithm [9]. CP records at each abstract state the current mapping from `AGMemLocs` to `AGValues`. Its transition function

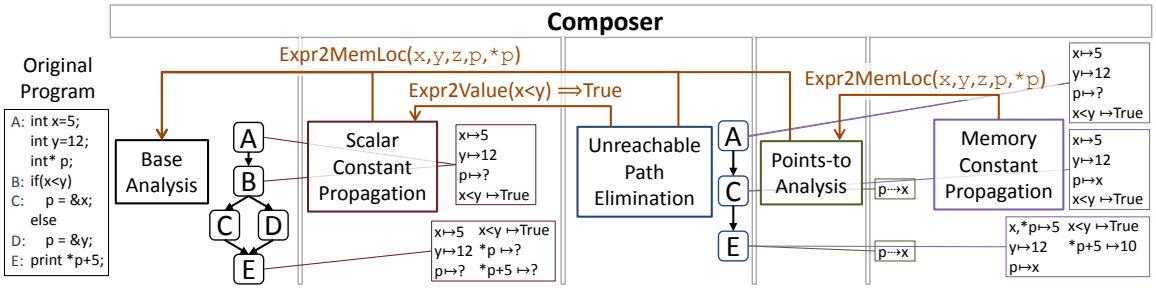

Fig. 10: Composition of Analyses in Fuse .

calls `Expr2MemLoc` to get the `AGMemLocs` for each expression's operands and updates their mappings. Since these calls are routed to Base, CP's mappings are accurate for lexical variables but not other locations (e.g. `*p`). CP implements only `Expr2Value`.

The next analysis is Unreachable Path Elimination (UPE). Since CP does not implement an abstraction graph, UPE runs on the one from Base. At the abstract node of the `if` conditional it calls `Expr2Value(x<y)` to get its result. The `AGValue` object `o` returned by CP denotes the set {True}, which is be accessed by calling `o.getConcrete`. UPE's abstraction graph implementation uses this to only generate nodes for the `if`'s true branch.

UPE is followed by a Points-To analysis (PT), which implements the Lhoták and Chong algorithm [11] and maintains a graph of points-to relationships among different memory locations. It calls `Expr2MemLoc` (routed to Base by the Composer) to get the `AGMemLocs` for the operands of each indirection (`&x`) or dereference (`*p`) operation and the graph it creates only maps the `AGMemLoc` for `p` to the `AGMemLoc` for `x`. It implements `Expr2MemLoc` to enable subsequent analyses to leverage this information.

Finally we run another instance of Constant Propagation (CP2), which has the functionality of the Wegman, Zadeck analysis [19] when combined with UPE. CP2's `Expr2MemLoc` calls are routed to PT, which resolves points-to relationships and returns the resulting `AGMemLoc` (returned objects are actually implemented by Base). Critically, the objects returned by `Expr2MemLoc(x)` and `Expr2MemLoc(*p)` are `mustEqual`. Thus, when CP2 reads the mapping of `*p` at the `print` operation `AGValue` that contains 5 (mapped to the `x` `AGMemLoc`) is returned. This allows CP2 to compute that 10 is printed.

8 Related Work

There is an abundance of work in the literature, including abstract interpretation, which bases static program analysis on the properties of abstract subexecution paths. Blanchet et al. [3] describe abstract domains, which are sets of abstract properties of execution traces and are implemented in ASTREE [5], a static program analysis tool. Rival, et al. [17] generalize abstract domains via

a generic trace partitioning abstract domain and define an abstract transition system similar to ours.

Lerner, et al. [10] present an analysis framework in which a super-analysis tightly composes multiple *integrated* analyses, i.e, ones in which analysis and transformation are integrated. They develop a mechanism for implicit communication between component analyses based on graph transformations. This mechanism limits client queries to expressions that occur in the program or in a transformed version of the program. Ramsey et al. [16] describe Hoopl, a polymorphic library that simplifies the addition of new dataflow analyses and transformations to compilers written in Haskell. Based on the work of Lerner, et al., Hoopl interleaves analysis with transformations. They develop a simple API for use with their library that has the same constraints as Lerner, et al.

Gulwani, et al. [7] present a method for combining abstract interpreters over multiple lattices to construct an abstract interpreter for their combination. They use a *logical product* of lattices, which refines a *reduced product* of lattices via the Nelson-Oppen [14] procedure. The approach in [7] focuses on interpreters that correspond to convex theories, allowing each interpreter to communicate must-equality constraints to other interpreters. Our interface supports may-equality constraints similar to inference on non-convex theories. Unlike [7], our framework allows the client to construct specific queries similar to ASTREE [5] ASTREE combines analyses based on communication between the abstract domains discussed above. However, their interface is based on writing special code to couple each pair of analyses.

Some domain specific languages for program analysis are well suited for composition. In Chord [1], analyses can be written declaratively in Datalog, a domain specific language for logical queries. Müller-Olm et al. [13] apply linear algebra techniques to determine affine and polynomial relations among program variables. Such frameworks can compose many analyses that fit within their constraints but do not apply to general analyses.

9 Conclusion

This paper proposes an approach to building compiler analysis frameworks that simplifies composition of independently-developed analyses by using transition systems as the underlying formalism. It defines concrete and abstract transition systems, and shows how the results of dataflow analyses can be used to specify abstractions that other analyses run on. Transitions systems are a portable abstraction that can represent the results of many real analyses, thereby making it possible for different analyses to leverage each other's results with no knowledge of their APIs or internal abstractions and without any coordination between the groups that developed them. We have developed the Fuse compositional analysis framework based on this abstraction, and integrated this framework in the ROSE [15] compilation system. Our experience is that this approach greatly simplifies composition of program analyses, thereby making it easy to tailor different combinations of program analyses to different applications.

References

1. Chord: A Program Analysis Platform for Java. http://pag.gatech.edu/chord/.
2. LLVM Alias Analysis Infrastructure. http://llvm.org/docs/AliasAnalysis.html.
3. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207, 2003.
4. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
5. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the astrée static analyzer. In *ASIAN*, pages 272–300, 2006.
6. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct 1991.
7. S. Gulwani and A. Tiwari. Combining abstract interpreters. In *PLDI*, pages 376–386, 2006.
8. M. Kelly and N. Spies. *Forth: A Text and Reference*. Prentice-Hall, 1986.
9. G. A. Kildall. A Unified Approach to Global Program Optimization. In *Symposium on Principles of Programming Languages (POPL)*, pages 194–206, New York, NY, USA, 1973. ACM.
10. S. Lerner, D. Grove, and C. Chambers. Composing Dataflow Analyses and Transformations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, pages 270–282, New York, NY, USA, 2002. ACM.
11. O. Lhoták and K.-C. A. Chung. Points-to Analysis with Efficient Strong Updates. In *Symposium on Principles of Programming Languages*, POPL '11, pages 3–16, New York, NY, USA, 2011. ACM.
12. T. Marlowe and B. Ryder. Properties of Dataflow Frameworks. *Acta Informatica*, 28:121–163, 1990.
13. M. Müller-Olm and H. Seidl. Precise Interprocedural Analysis Through Linear Algebra. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '04, pages 330–341, New York, NY, USA, 2004. ACM.
14. G. Nelson and D. C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, Oct. 1979.
15. D. Quinlan. ROSE: Compiler Support for Object-Oriented Frameworks. In *Conference on Parallel Compilers (CPC)*, 2000.
16. N. Ramsey, J. Dias, and S. L. P. Jones. Hoopl: a modular, reusable library for dataflow analysis and transformation. In *Haskell*, pages 121–134, 2010.
17. X. Rival and L. Mauborgne. The Trace Partitioning Abstract Domain. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(5), Aug. 2007.
18. D. A. Schmidt and B. Steffen. Program Analysis as Model Checking of Abstract Interpretations. 1503:351–380, Sept. 1998.

19. M. N. Wegman and F. K. Zadeck. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.